Date=27/08/2020
Lecture By=Shubham Joshi
Subject ⇒ Graphs

IN PREVIOUS LECTURE **(QUICK RECAP) Date-26/08/2020**	**In Today's Lecture (Overview)**
Kadane's Algorithm: Question=1⇒ Largest Sum Contiguous Subarray Write an efficient program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum Mcqs Questions For self Practice // Assignment For the Day	Graphs in python Types of graphs ⇒ Directed graphs ⇒ Undirected Graphs ⇒ Weighted graphs ⇒ Unweighted graphs ⇒ MCQs

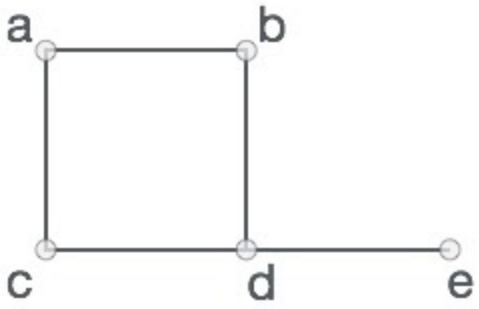
In Today's lecture we discussed About the graphs in python So in this notes i am gonna discussed about the same

Graphs in python

First of all let's know learn about graphs
A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

A graph can be easily presented using the python dictionary data types. We represent the vertices as the keys of the dictionary and the connection between the vertices also called edges as the values in the dictionary.

Take a look at the following graph −

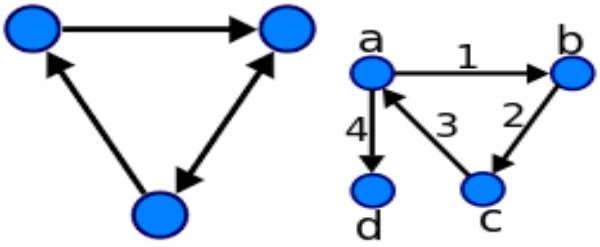


Types of graphs

These are the types of the graphs

⇒ Directed graphs

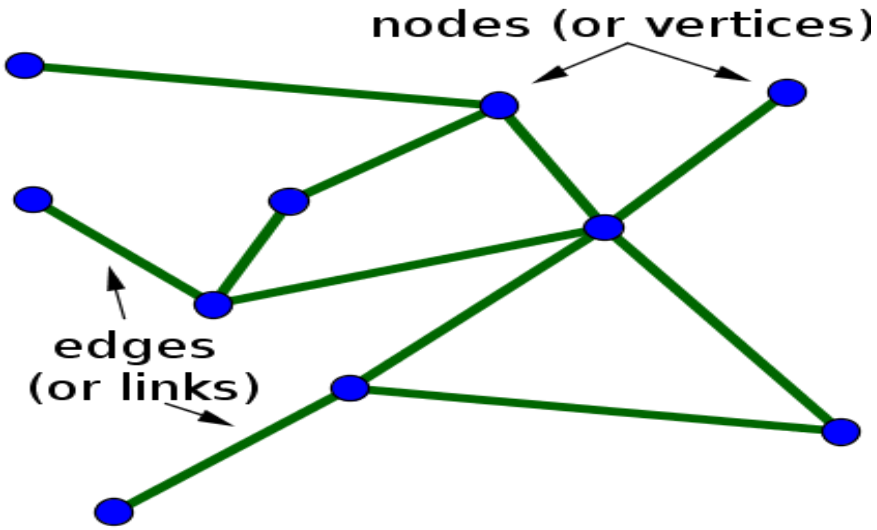
=A **directed graph** is a graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are **directed** from one vertex to another. A **directed graph** is sometimes called a digraph or a **directed** network.



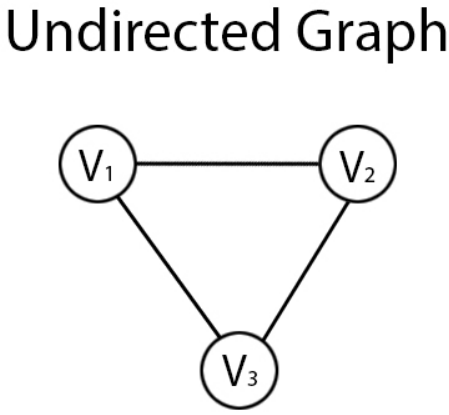
	1	2	3	4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1

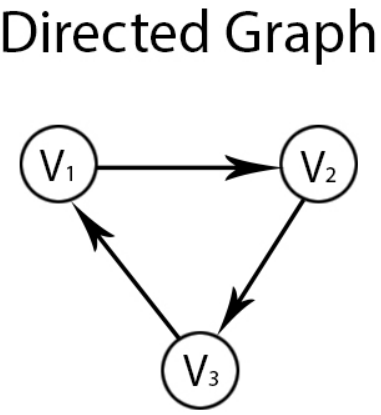
⇒ Undirected Graphs

An **undirected graph** is a graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An **undirected graph** is sometimes called an **undirected** network. In contrast, a **graph** where the edges point in a direction is called a directed **graph**.



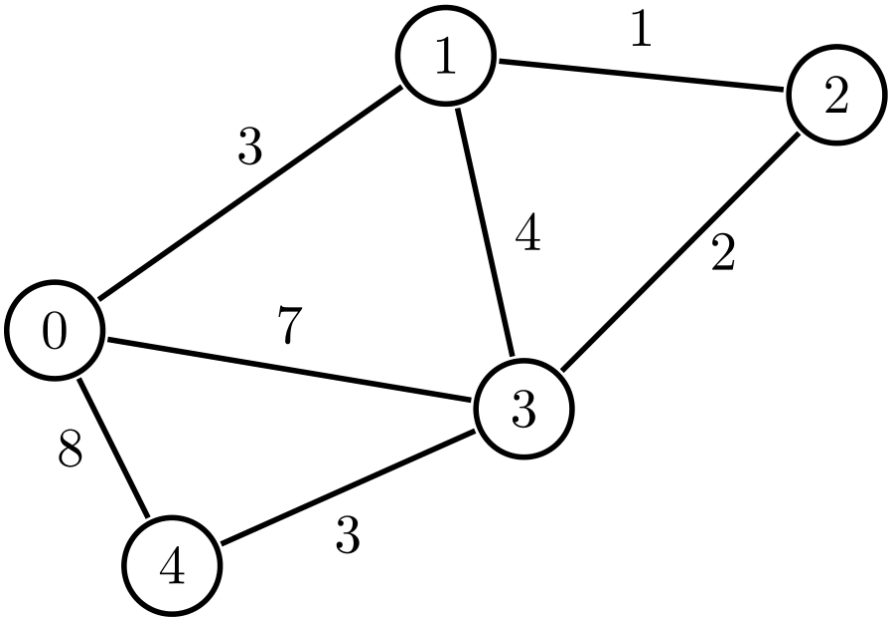
Difference Between Both Types





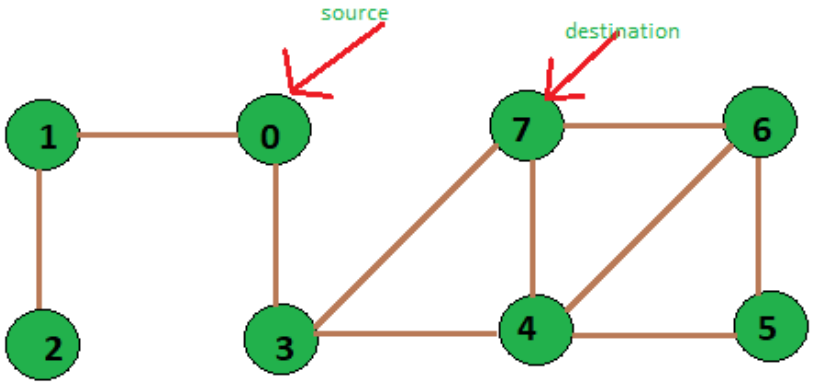
⇒ Weighted graphs

A **weighted graph** is a **graph** in which each branch is given a numerical weight. A **weighted graph** is therefore a special type of labeled **graph** in which the labels are numbers (which are usually taken to be positive).



⇒ Unweighted graphs

If edges in your **graph** have weights then your **graph** is said to be a weighted **graph**, if the edges do not have weights, the **graph** is said to be **unweighted**. A weight is a numerical value attached to each individual edge.

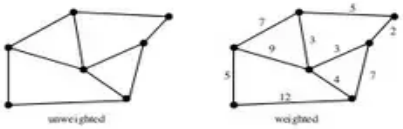


“Click here”to know more about it.

Difference between both

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.



The edges of a road network graph might be weighted with their length, drive-time or speed limit.
In *unweighted* graphs, there is no cost distinction between various edges and vertices.

Now that you know about the graphs lets discussed about the implementation of the code

Code for implementing the graph in python

```
# import dictionary for graph
from collections import defaultdict

# function for adding edge to graph
graph = defaultdict(list)
def addEdge(graph,u,v):
    graph[u].append(v)

# definition of function
def generate_edges(graph):
    edges = []

    # for each node in graph
    for node in graph:
```

```
        # for each neighbour node of a single node
        for neighbour in graph[node]:

            # if edge exists then append
            edges.append((node, neighbour))
    return edges

# declaration of graph as dictionary
addEdge(graph,'a','c')
addEdge(graph,'b','c')
addEdge(graph,'b','e')
addEdge(graph,'c','d')
addEdge(graph,'c','e')
addEdge(graph,'c','a')
addEdge(graph,'c','b')
addEdge(graph,'e','b')
addEdge(graph,'d','c')
addEdge(graph,'e','c')

# Driver Function call
# to print generated graph
print(generate_edges(graph))
```

#this code is from geeks for geeks and is just for example

This is the code that was discussed in The Lecture

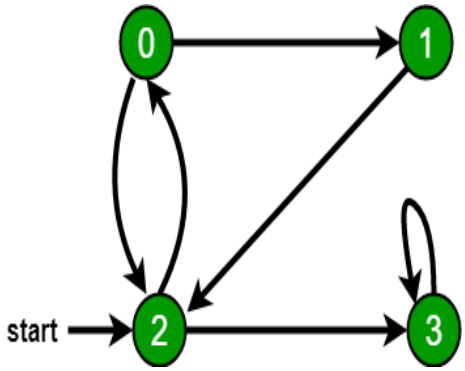
```
# Adjacency List


def addEdge(u, v, graph, undirected=True):
    if u not in graph:
        graph[u] = list()
    graph[u].append(v)

    if undirected:
        if v not in graph:
            graph[v] = list()
        graph[v].append(u)
```

```python
def addEdgeWithWeight(u, v, weight, graph, undirected=True):
    if u not in graph:
        graph[u] = list()
    graph[u].append((v, weight))

    if undirected:
        if v not in graph:
            graph[v] = list()
            graph[v].append((u, weight))


# Adjacency Matrix representation

def addEdgeAdjMatrix(u, v, graph, undirected=True):
    graph[u][v] = 1
    if undirected:
        graph[v][u] = 1

def addEdgeAdjMatrixWeighted(u, v, weight, graph, undirected=True):
    graph[u][v] = weight
    if undirected:
        graph[v][u] = weight


if __name__ == '__main__':
    # graph = dict()
    # addEdge(1, 2, graph, False)

    graph = [[0 for x in range(100)] for x in range(100)]
    addEdgeAdjMatrix(0, 4, graph)
```

Bfs Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree . The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Code for Bfs Implement

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)
```

```
    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code

# Create a graph given in
```

```
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
```

⇒ MCQs

1.**Is the adjacency list correct ?**

A=no

B=yes

2.**can a graph have no edges ?**

A=No never

B=Yes can be

C=Yes might be

D=No in some cases only

3.**Is Tree a graph ?**

A=No

B=Yes